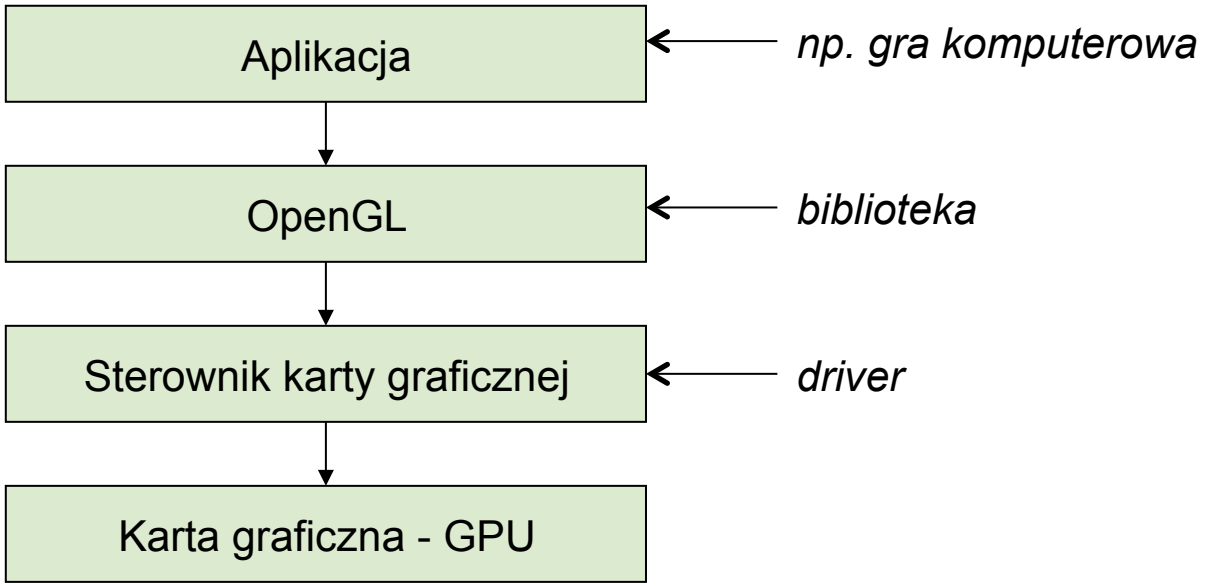
Biblioteka OpenGL: Wprowadzenie

Radosław Mantiuk

Wydział Informatyki
Zachodniopomorski Uniwersytet Technologiczny w Szczecinie

OpenGL - Koncepcja i architektura

Aplikacja odwołuje się poprzez funkcje API OpenGL bezpośrednio do karty graficznej (z pominięciem systemu operacyjnego).

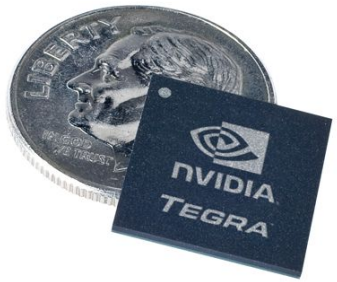


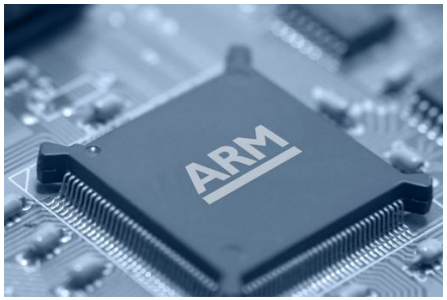
Implementacja software'owa - wszystkie funkcje mogą być wykonywane przez CPU, jednak producenci kart mogą zastępować część funkcji OpenGL swoimi rozwiązaniami sprzętowymi. Dla programisty ta zamiana jest niewidoczna.

Procesory GPU (Graphics Processor Unit)

Programowalny procesor pracujący niezależnie od CPU.

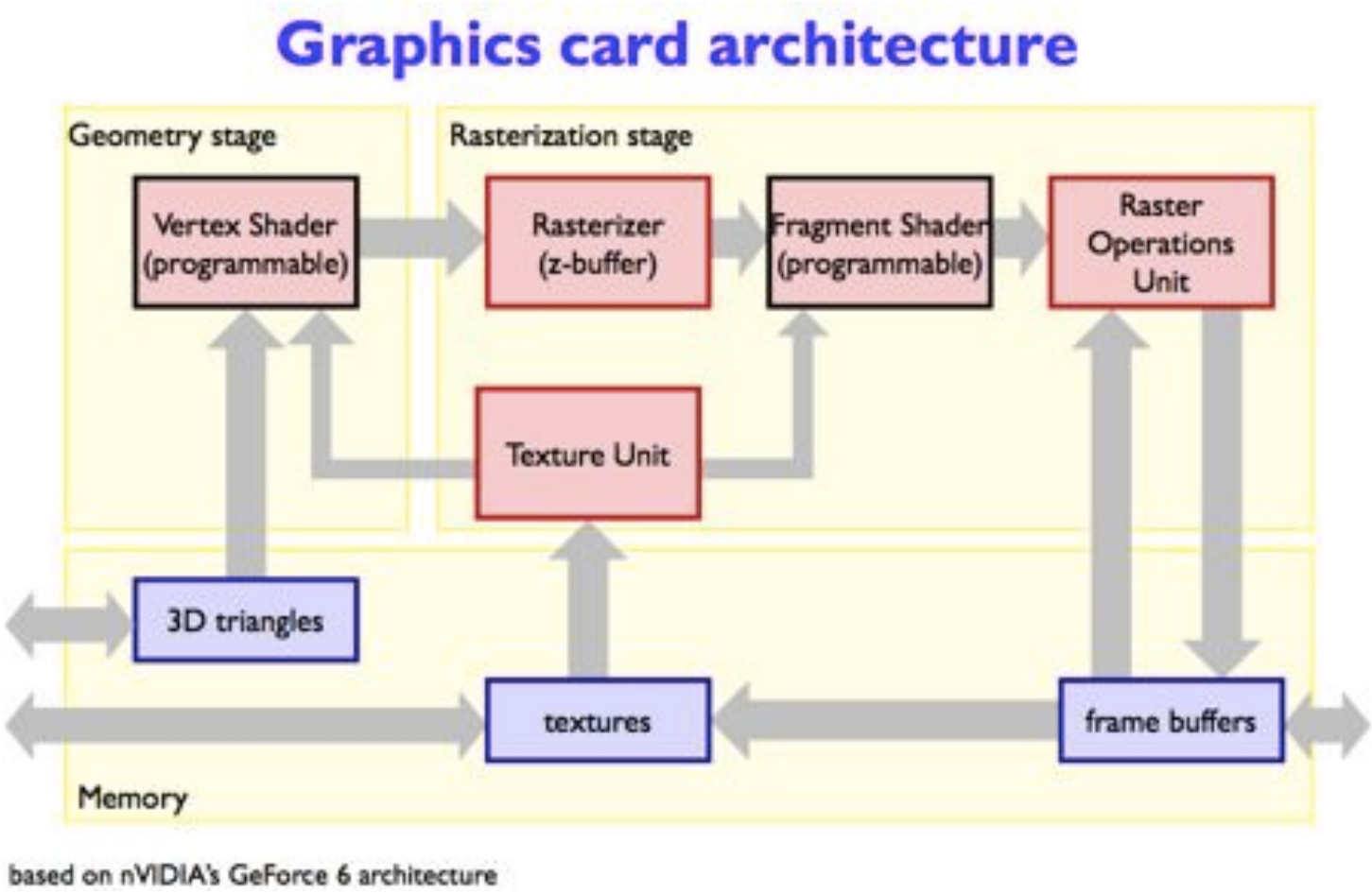
- szybkie operacje macierzowe - architektura SIMD (w ograniczonym zakresie MIMD),
- zrównoleglenie obliczeń (wiele potoków renderujących, stream programming),
- mogą być wykorzystane do innych obliczeń GPGPU (General Purpose GPU),
- szybszy rozwój niż CPU (brak rozbudowanej logiki).





Architektura GPU

Programowalny procesor pracujący niezależnie od CPU.



OpenGL - Open Graphics Library

- **OpenGL: architektura systemu do programowania grafiki komputerowej (programowanie grafiki -> synteza obrazów)**
  - Wspomaganie sprzętowe (system graficzny czasu rzeczywistego).
  - Bazuje na specyfikacji (nie implementacji).
  - Rozszerzalność.
  - Wieloplatformowość (OpenGL ES, wiele typów danych).
  - Możliwość połączenia z różnymi językami programowania.
  - Maszyna stanu (ang. rendering state machine).

- 1993: pierwsza wersja OpenGL 1.0 (== IRIX GL 5.0) w 1993 roku (IRIX GL 1.0 1983).
- 2001: OpenGL 1.4, vertex shader (programowalość operacji na wierzchołkach).
- 2003: OpenGL 1.5, fragment shader (oprogramowalność operacji na fragmentach).
- Standard - OpenGL Khronos Group (ARB (Architecture Review Board), 3Dlabs, Apple, ATI, Dell Computer, IBM, Intel, NVIDIA, SGI, and Sun Microsystems).
- Implementowana przez producentów kart graficznych.
- Dostępna na większości platform sprzętowych i programowych (OpenGL ES).
- OpenGL 4.2 (sierpień 2011)

OpenGL: Wieloplatformowość

Biblioteka OpenGL nie obsługuje operacji wejścia (np. myszy, klawiatury) i wyjścia (np. monitora).

GLFW

Przykład interfejsu pomiędzy OpenGLem i systemem operacyjnym.

Funkcje:

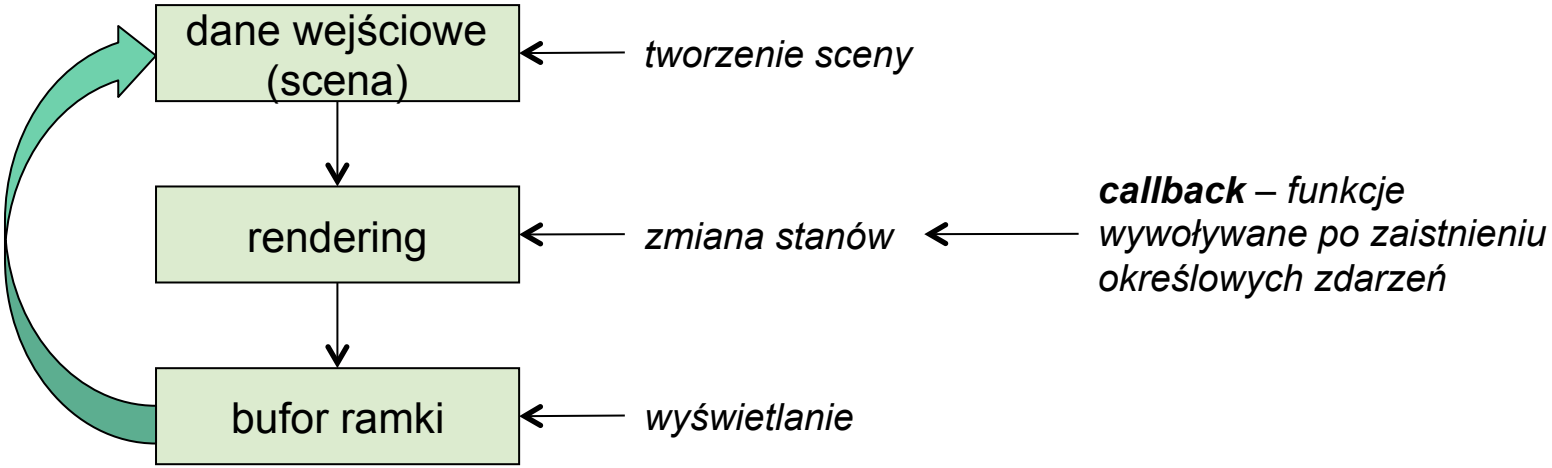
- Wyświetlanie okien, w których może odbywać się renderowanie realizowane przez OpenGL (może być otwarte wiele takich okien),
- Obsługa zdarzeń wejścia/wyjścia za pomocą callback'ów.
- Obsługa urządzeń wejściowych (klawiatura, mysz, joystick).
- Obsługa timer'ów.

Inne biblioteki: **GLUT, FreeGLUT, SDL**.

OpenGL: Maszyna stanu

OpenGL to zestaw funkcji służących do renderowania obrazu do bufora ramki.

Na wyniki renderingu można wpływać poprzez zmianę stanów.



Przykład aplikacji: OpenGL + GLUT

```
int main(int argc, char* argv[]) {

        glutInit(&argc, argv);
        glutInitWindowSize(1024,1024);
        glutInitDisplayMode(GLUT_DOUBLE | GLUT_DEPTH | GLUT_RGBA);
        glutInitWindowPosition(10,100);
        glutCreateWindow("GL Game");

        glutDisplayFunc(display);
        glutKeyboardFunc(keyboard_press);
        glutKeyboardUpFunc(keyboard_up);
        glutTimerFunc( 1000, timer, 0 );

        glutMainLoop();

        return 0;
}
```

```
void display(void) {
// wyczyszczenie bufora ramki (ang. frame buffer)
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// kolor obiektu
glMaterialfv( GL_FRONT, GL_DIFFUSE, colorGreen );
glMaterialfv( GL_FRONT, GL_AMBIENT, colorDim );
glMaterialfv( GL_FRONT, GL_EMISSION, colorDim );
glMaterialfv( GL_FRONT, GL_SPECULAR, colorDim );

// polozenie i ksztalt obiektu
glPushMatrix();
glRotatef( -30.0f, 0.0f, 1.0f, 0.0f );
glutSolidSphere( 1.5, 30, 30 );
glPopMatrix();

// wyswietlenie tylniego ekranu
glutSwapBuffers();

// wymyszenie przerysowania podczas nastepnej synchronizacji ekranu
glutPostRedisplay();
}
```

OpenGL: Potok przetwarzania (ang. pipeline)

potok – ciąg operacji wykonywanych przez silnik renderujący. Przetwarzanie trójkątów od reprezentacji 3D do pikseli w buforze ramki.

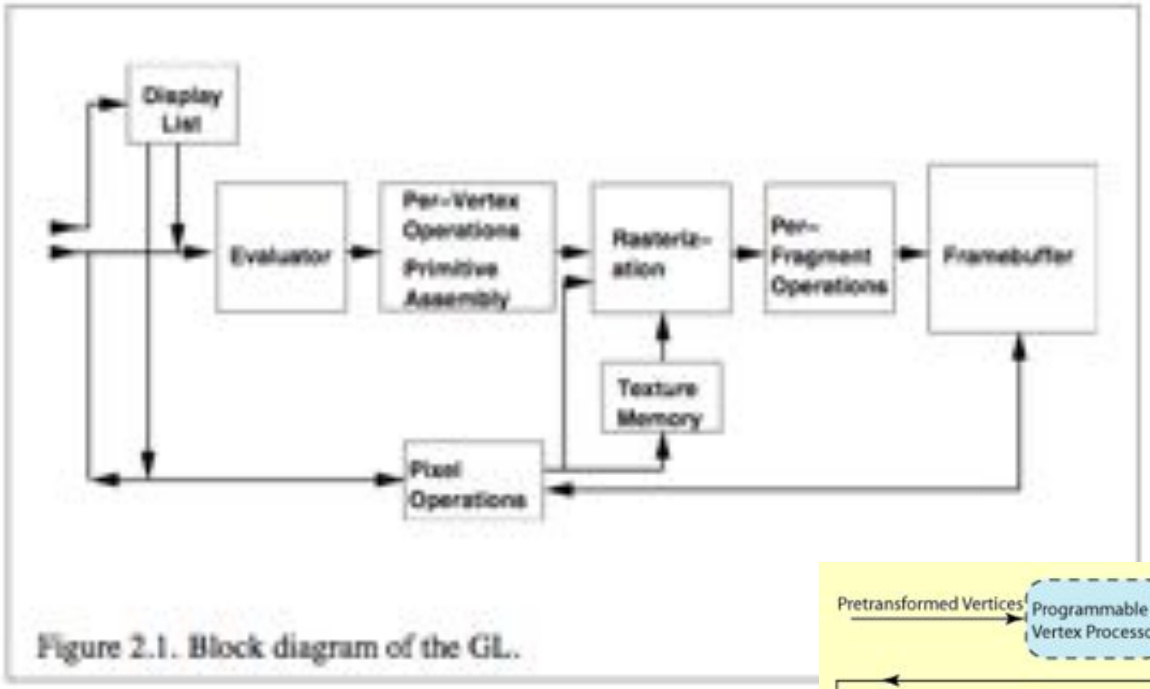
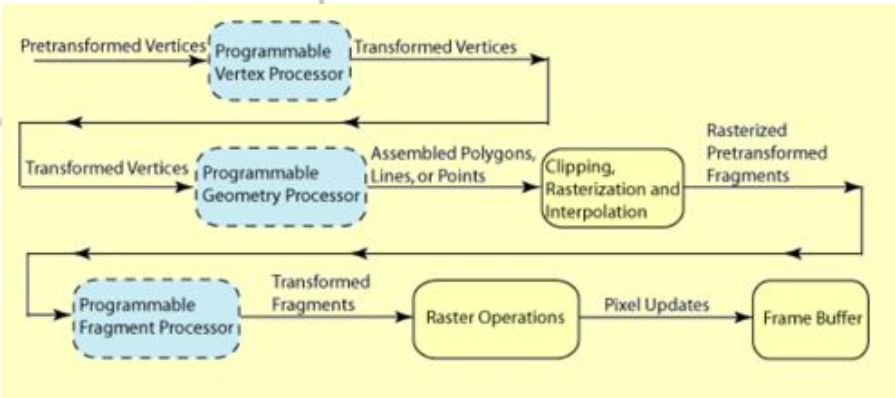


Figure 2.1. Block diagram of the GL.

Shader Model 4.0



fixed pipeline vs. programmable pipeline

Shaders

shader – program uruchamiany na karcie graficznej i wykonywany przez GPU

- kompilacja run-time
- składnia uwzględniająca przetwarzanie wektorowe
- pass-through lub zmiana potoku renderingu
- vertex shader, geometry shader, fragment shader
- wykonywane równolegle przez "rdzenie" GPU

- **jeden kod programu wykonywany jest dla wielu danych (np. wszystkich wierzchołków modelu)**

Vertex shader

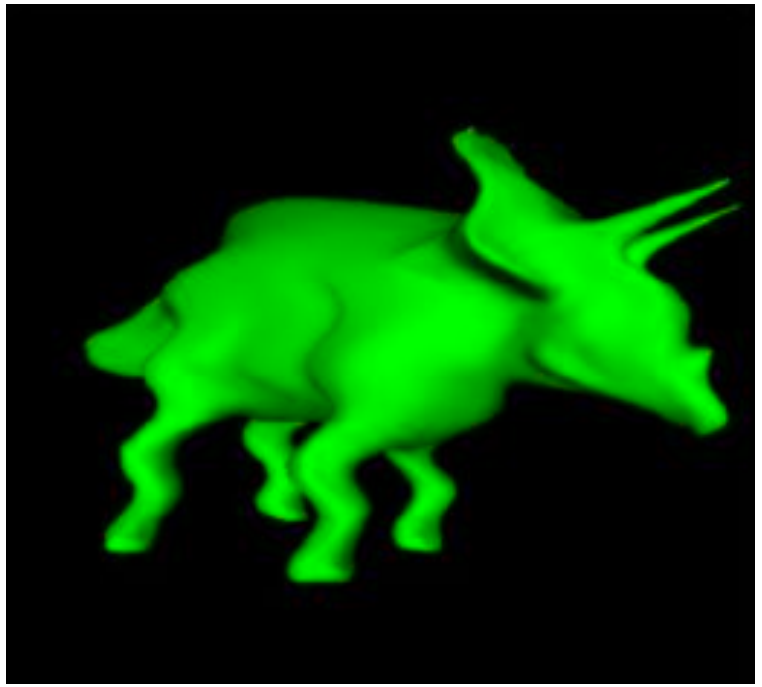
niejednorodna mgła, której występowanie uzależnione jest od współrzędnych wierzchołków

kaustyki utworzone poprzez modyfikacje kształtu powierzchni wody

Vertex shader

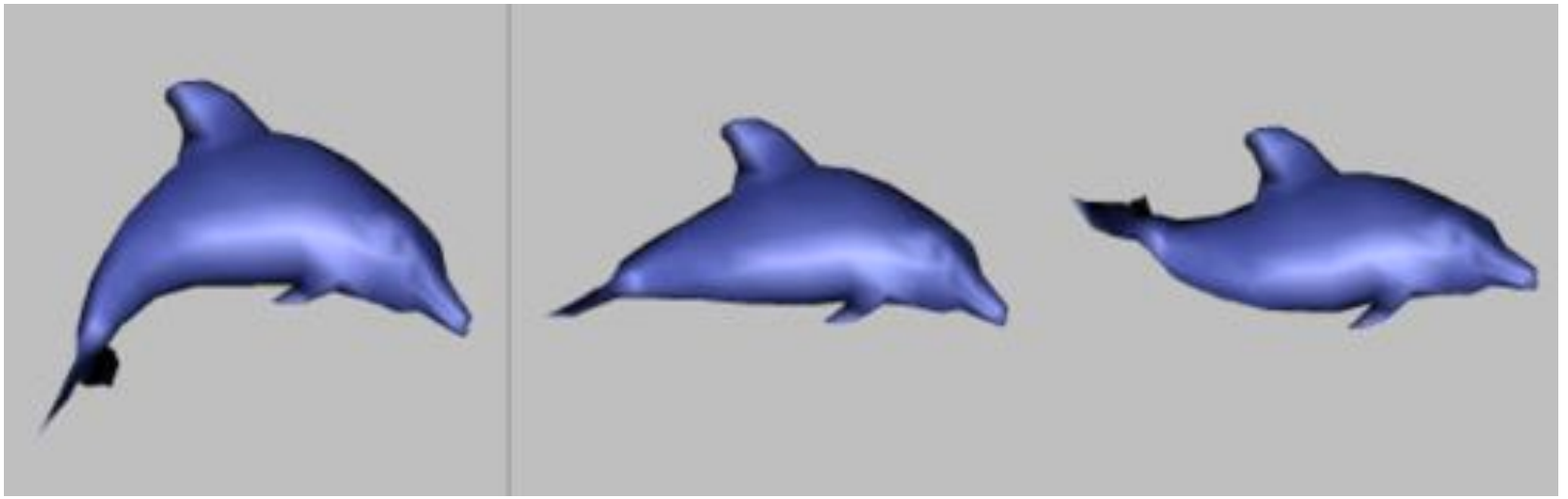
Program przetwarzający wierzchołki.

zmiana mimiki twarzy, wspomaganie animacji postaci

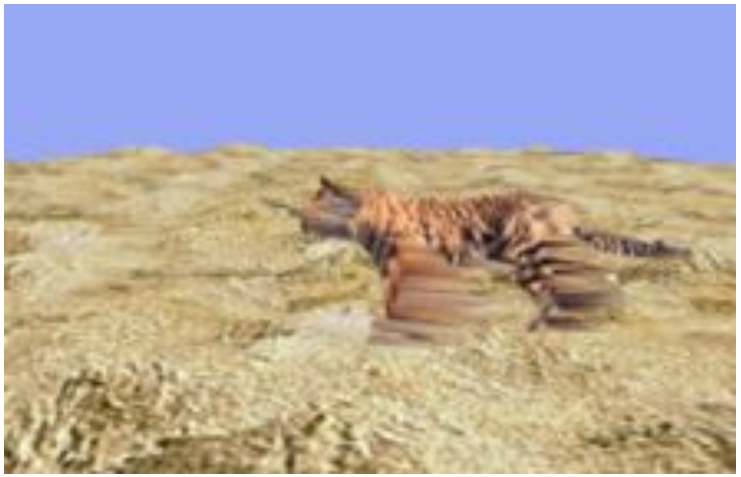


deformacje proceduralne

Vertex shader



morphing

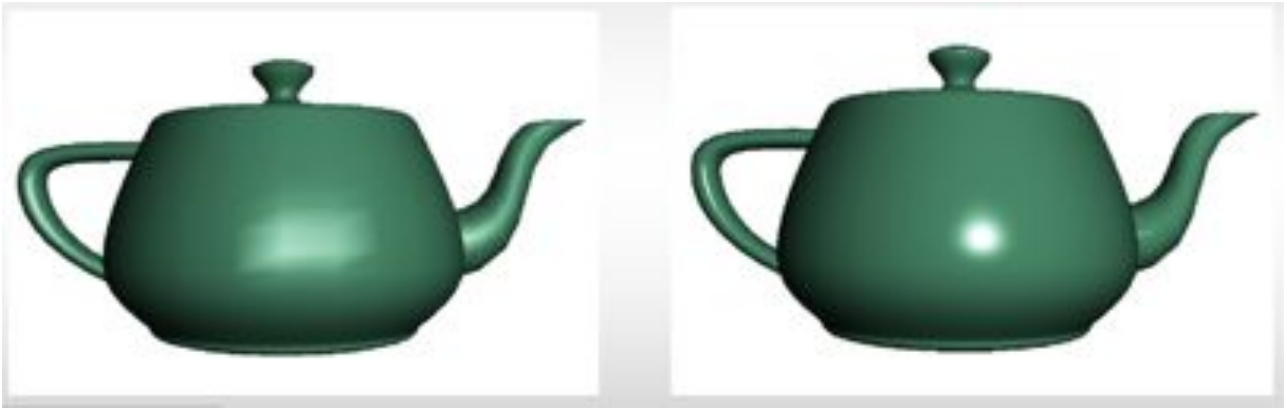


rozmycie ruchu

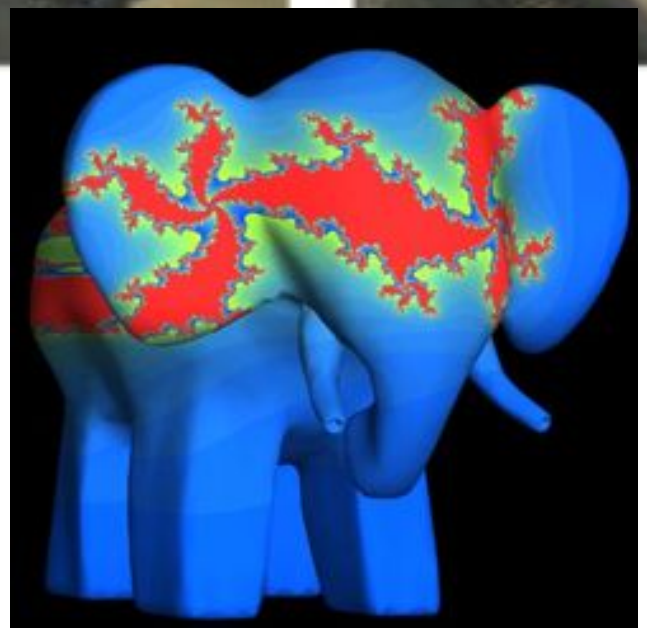
Fragment shader

Transformacje związane z fragmentami:
• obliczanie koloru fragmentu,
• teksturowanie,
• obliczanie oświetlenia (Phong shading),
• mgła.

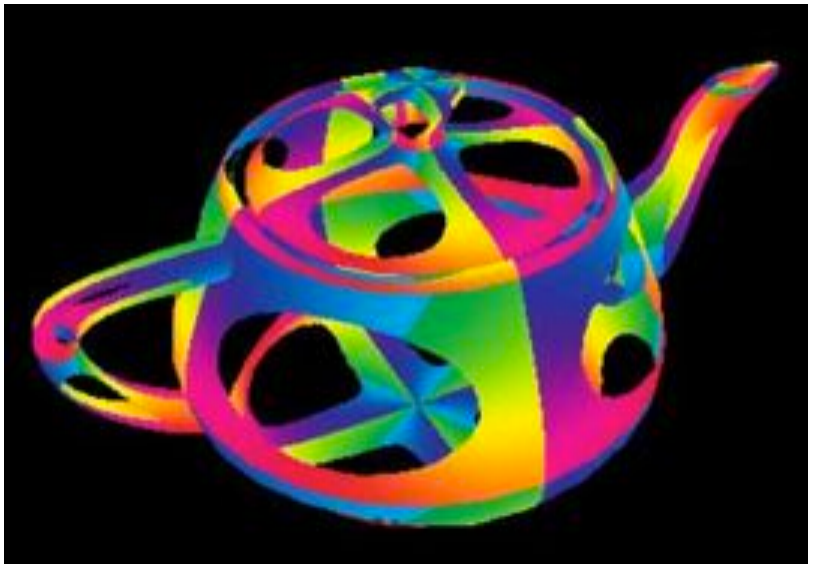
Zastosowania:
• proceduralne teksturowanie,
• obliczanie tekstury z wykorzystaniem danych ze standardowych tekstur,
• mapowanie tekstur.



Fragment shader - przykłady

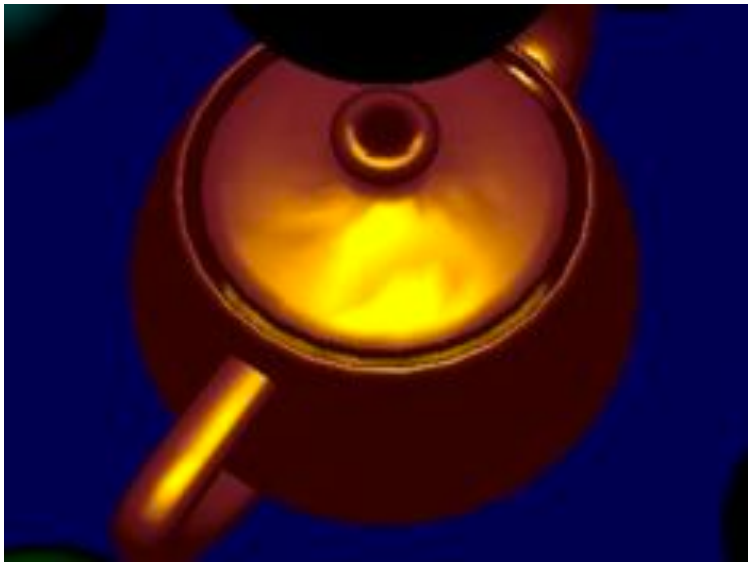


Fragment shader - przykłady



przezroczystość

grafika nierealistyczna



bump mapping

Programowanie shaderów

Języki programowania:
- GLSL - OpenGL Shading Language,
- CG (nVidia)
- HLSL (Microsoft)

```
/* Read shaders into the appropriate buffers */
   vertexsource = CUTIL::filetobuf("phong.vert");
   fragmentsource = CUTIL::filetobuf("phong.frag");

/* Create an empty vertex shader handle */
   vertexshader = glCreateShader(GL_VERTEX_SHADER);

/* Send the vertex shader source code to GL */
glShaderSource(vertexshader, 1, (const GLchar**)&vertexsource, 0);

/* Compile the vertex shader */
 glCompileShader(vertexshader);

/* Create an empty fragment shader handle */
 fragmentshader = glCreateShader(GL_FRAGMENT_SHADER);

/* Send the fragment shader source code to GL */
glShaderSource(fragmentshader, 1, (const GLchar**)&fragmentsource, 0);

 /* Compile the fragment shader */
glCompileShader(fragmentshader);

/* Linking */
shaderprogram = glCreateProgram();

/* Attach shaders to the program */
glAttachShader(shaderprogram, vertexshader);
glAttachShader(shaderprogram, fragmentshader);

glLinkProgram(shaderprogram);

glUseProgram(shaderprogram);
```

Vertex shader - Phong shading

```
varying vec3 N;
varying vec3 v;

void main(void) {

        v = vec3(gl_ModelViewMatrix * gl_Vertex);
        N = normalize(gl_NormalMatrix * gl_Normal);

        gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

}
```

Fragment shader - Phong shading

```
varying vec3 N;
varying vec3 v;

void main (void) {

vec4 Iamb; vec4 Idiff; vec4 Ispec;

for( int i = 0; i<3; i++ ) {

        vec3 L = normalize(gl_LightSource[i].position.xyz - v);
        vec3 E = normalize(-v);
        vec3 R = normalize(-reflect(L,N));

        //calculate Ambient Term:
        Iamb = Iamb  + gl_FrontLightProduct[i].ambient;

        //calculate Diffuse Term:
        Idiff = Idiff + gl_FrontLightProduct[i].diffuse * max(dot(N,L), 0.0);

        // calculate Specular Term:
        Ispec = Ispec + gl_FrontLightProduct[i].specular * pow(max(dot(R,E),0.0),
        gl_FrontMaterial.shininess);
}
// write Total Color:
gl_FragColor = gl_FrontLightModelProduct.sceneColor + Iamb + Idiff + Ispec ;

}
```

Vertex Buffer Object (VBO)

Zestaw funkcji umożliwiających wczytanie danych o geometrii do pamięci karty graficznej.

Utworzenie bufora VBO:
GenBuffers(sizei n, uint *buffers)

Aktywacja VBO:
BindBuffer(enum target, uint buffer)

Wczytanie danych do VBO:
BufferData(enum target, sizeiptrARB size, const void *data, enum usage)

Skasowanie VBO:
DeleteBuffers(sizei n, const uint *buffers)

Vertex Buffer Object (VBO)

```
/*Initialise VBO - (Note: do only once, at start of program)*/

/* Create a new VBO and use the variable "triangleVBO" to store the VBO id */
GLuint triangleVBO;
glGenBuffers(1, &triangleVBO);

/* Make the new VBO active */
glBindBuffer(GL_ARRAY_BUFFER, triangleVBO);

/* Upload vertex data to the video device */
glBufferData(GL_ARRAY_BUFFER, NUM_OF_VERTICES_IN_DATA * 3 * sizeof(float), data,
GL_STATIC_DRAW);

/* Specify that our coordinate data is going into attribute index 0(shaderAttribute), and
contains three floats per vertex */
glVertexAttribPointer(shaderAttribute, 3, GL_FLOAT, GL_FALSE, 0, 0);

/* Enable attribute index 0(shaderAttribute) as being used */
glEnableVertexAttribArray(shaderAttribute);

/* Make the new VBO active. */
glBindBuffer(GL_ARRAY_BUFFER, triangleVBO);
```

OpenGL Mathematics (GLM Library)

Biblioteka operacji matematycznych wykorzystywanych w języku GLSL.
http://www.opengl.org/sdk/libs/GLM/
http://glm.g-truc.net/0.9.5/index.html

```
#include <glm/glm.hpp>

int foo() {
     glm::vec4 Position = glm::vec4(glm::vec3(0.0), 1.0);
     glm::mat4 Model = glm::mat4(1.0);
     Model[3] = glm::vec4(1.0, 1.0, 0.0, 1.0);
     glm::vec4 Transformed = Model * Position;
     return 0;
}
```

OpenCL: Open Computing Language

środowisko wspierająca pisanie oprogramowania, które będzie uruchamiane na niejednorodnych platformach sprzętowych (CPU, GPU, FPGA, DSP):
- interfejs kontrolujący platformy sprzętowe (w tym uruchamiający ***kernele*** na urządzeniach)
- jednorodny język programowania różnych urządzeń,
- interfejs do obliczeń równoległych,
- zaawansowane zarządzanie pamięcią (4 poziomy dostępu)

Cechy techniczne:
- język podobny do C
- wspomaganie obliczeń wektorowo-macierzowych

Khronos – standard (OpenCL 2.1)
AMD – od sierpnia 2015 pełna implementacja OpenCL 2.0

Literatura

- Dave Shreiner, The Khronos OpenGL ARB Working Group, Bill Licea-Kane, Graham Sellers, **OpenGL Programming Guide: The Official Guide to Learning OpenGL**, Addison-Wesley Professional, USA, 2012